Computer Physics Communications 60 (1990) 329–344
North-Holland

A review of pseudorandom number generators

F. James
CERN, CH-1211 Geneva 23, Switzerland

Received 2 May 1990

This is a brief review of the current situation concerning practical pseudorandom number generation for Monte Carlo calculations. The conclusion is that pseudorandom number generators with the required properties are now available, but the generators actually used are often not good enough. Portable Fortran code is given for three different pseudorandom number generators, all of which have much better properties than any of the traditional generators commonly supplied in most program libraries.

PROGRAM SUMMARY

Title of program: PSEUDORAN

Catalogue number: ABTK

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Computer for which the program is designed: It should give the same results on any computer with a Fortran 77 compiler, and has been tested on Apollo 3500 and Apollo 10000 (RISC) under Sr10 Unix, Cray X-MP under Unicos, IBM 3090 under VM/CMS/XA, and Vax 8600 under VMS

Programming language used: Fortran 77

No. of lines in combined program and test deck: 367

Keywords: random numbers, pseudorandom, uniform distribution, long period, independent subsequences

Nature of physical problem
Any Monte Carlo calculation requiring uniform pseudorandom numbers.

Method of solution
Three different methods are proposed, each of which gives independent pseudorandom number sequences with very good properties.

Restrictions on the complexity of the problem
The generators proposed here all produce 32-bit floating-point numbers uniform between zero and one. The simplest generator proposed has a period of about 10^{18}, which could conceivably be too short for some exceptional calculations, but the others have much longer periods.

Typical running time
The speed of these generators is comparable to, and in some cases faster, than traditional hand-coded generators. Timings are given in the long write-up of the article. They could of course be speeded up further at the cost of making them nonportable.

Unusual features of the program
These generators are completely portable (both the code and the numbers produced), and are suitable for parallel and vector processing as well as traditional applications.

LONG WRITE-UP

1. General considerations

1.1. The motivation and scope of this paper

Physicists often need very good random number generators for Monte Carlo calculations, but seldom feel the need to spend any considerable effort to assure themselves of the quality of the generators they use. Unfortunately, even though the generators available at most computer centres were considered good when they were first installed, it is unlikely that a generator of twenty or even ten years ago will continue to meet the requirements of today's increasingly long and complex calculations. Indeed the history of Monte Carlo computation has been marked, from the earliest days right up to the present, by physicists making the painful discovery that their calculations are unreliable because the local random number generator did not have the properties required of it. I will spare many readers from unpleasant recollections by not citing the many internal reports devoted to the revelation that the local "official random number generator" is not random enough.

Recent progress in both theoretical understanding and practical implementation of random number generators has been such that it is now possible to find an appropriate generator available off the shelf for nearly any practical computation. The purpose of this review is to summarize the requirements for a good generator, and to give examples of generators fulfilling these requirements. It is up to the individual physicists to make sure that the one they use (and preferably the default at their computer centre) is appropriate for their calculations.

Only uniform generators are covered here. There exist in addition an enormous number of techniques for generating random numbers according to other distributions, for example Gaussian, Poisson, binomial, exponential, etc. Most of these techniques require a good uniform generator underneath.

1.2. The three types of generators

Random number generators for Monte Carlo calculations can be classified according to the three types of numbers generated:

- Truly random numbers are unpredictable in advance and must be produced by a random physical process, such as radioactive decay. Series of such numbers are available on magnetic tape or published in books, but they are extremely clumsy to use and are generally insufficient in both number and accuracy for serious calculations. They are not discussed here any further.
- Pseudorandom numbers are produced in the computer by a simple numerical algorithm, and are therefore not truly random, but any given sequence of pseudorandom numbers is supposed to appear random to someone who doesn't know the algorithm. These are the most commonly used and are discussed below.
- Quasirandom numbers are also produced by a numerical algorithm, but are not designed to appear to be random, but rather to be distributed as uniformly as possible, in order to reduce the errors in Monte Carlo integration. They are not appropriate for all Monte Carlo calculations, but can be highly advantageous where appropriate. These are not discussed in this paper, but will be the subject of a future paper.

1.3. Desirable properties of a random number generator

The first property given below (good distribution) is important for all calculations. The other properties are not always needed, but a good general purpose generator should possess them all.

1. *Good distribution*. For pseudorandom numbers, this means good *randomness*. For quasirandom numbers, the desired quality is *uniformity*. The exact meaning of these terms in the context of this paper is discussed below.

2. *Long period*. Both pseudorandom and quasirandom generators always have a period, after which they begin to generate the same sequence of numbers over again. In any particular calculation, it is dangerous to come anywhere near exhausting the period, especially for pseudorandom generators. In the early days it was believed that a long period was sufficient also to guarantee a good distribution, but this is now known not to be true in general. Traditional pseudorandom generators are based on a single integer "seed", which means that the period is limited to the number of different states that can be represented in one computer word. For practical reasons, two bits are usually lost (for positivity and to avoid even integers), so for a 32 bit computer a simple generator can have a maximum period of 2^{30} or about 10^9. Although it is easy to achieve this maximum, it is no longer enough for many present day problems. Traditional methods can be extended, even on 32-bit computers, to give periods equal to the number of states representable in 60 bits. Some modern methods have periods much longer than 2^{60}; these methods will be referred to as VLP (very long period) methods.

3. *Repeatability*. There are really two aspects:

- For testing and development, it may be necessary to repeat a calculation with exactly the same random numbers as in the previous run. Nearly all generators do this by default, the only exception being those few that introduce an element of true randomness by initializing to an external device like the system clock.
- A somewhat more tricky case, again arising in program testing, is to be able to repeat part of a job without redoing the whole thing. For example, in event simulation, if event number 368 provokes an exception, one wants to be able to study it again without regenerating the first 367 events. As a general rule, this requires recording the state of the generator at the beginning of each event, which for simple generators means only remembering one number, the seed. More complex generators require storing a larger amount of information; one of the generators recommended below needs a vector of 100 fullword numbers to define its state at any moment, although only one 32-bit integer is required to initialize it.

4. *Long disjoint subsequences*. For large problems, particularly those being solved by a large team of physicists simultaneously, it is extremely convenient to be able to perform independent subsimulations whose results can later be combined assuming statistical independence. None of the traditional generators allows this to be done conveniently. The traditional technique for continuing a calculation is to record the value of the "seed", giving the current state of the generator, at the end of each subcalculation, and feed that into the next subcalculation to restart the generator where it left off. This technique does not allow one subcalculation to start before the previous one has finished, and also requires considerable inconvenient bookkeeping and perhaps interteam communication. Some of the generators proposed below solve this problem.

5. *Portability*. This means not only that the code should be portable (i.e. in a high-level language like Fortran), but that it should generate exactly the same sequence of numbers on different machines, in order to verify that the programs give the same results on different machines, at least to within the machine accuracy. Of course, certain kinds of calculations, in particular those involving branches which depend on the results of floating-point calculations, may give very different results due to different hardware arithmetic accuracy, even if the random numbers are rigorously identical, so the comparison may not work for programs of this kind.

6. *Efficiency*. This was considered very important in the early days, but with the kind of computations being performed now, both the computer time and memory space taken by random number generation are increasingly insignificant and can almost always be neglected. In fact, the way random number generators are traditionally implemented, as Fortran functions returning one random number per call, the CPU time is usually dominated by the time to make the function call, so the actual calculation time for the generation is seldom of any importance.

If CPU time for random number generation is a problem, there are only two ways to get around it: (1) by coding the random number generator inline, or (2) by implementing the generator as a subroutine which returns an array of random numbers at each call. Method (1) may be worthwhile in specialized applications, but may not be convenient in big programs where the random generator is called from many places. Method (2) is always to be recommended since the penalty to the user who can handle only one number at a time is small compared with the gain to the clever user.

Moreover, it is important to note that *in vector computers, even if the time spent in the generator is negligible, the above techniques can still result in enormous overall timing improvements, since otherwise a call to the random generator in a loop will prevent any vectorization of the loop.*

1.4. Manufacturer-supplied generators

It has been traditional to use pseudorandom number generators supplied by the manufacturers of the local computer. There are many reasons for this, but most of them are no longer valid. The main reason was probably blind faith in the superior technical expertise of the manufacturer and the belief that a good generator must be written in assembler and exploit particular hardware features of the machine in order to be efficient. Many users are also influenced by the vague feeling that a sequence can be random only if it is produced in a mysterious way, coded in an unknown language whose source is not available.

Given the current state of the art, there is practically no reason to use manufacturer's software for random number generation, the possible exception being those (supercomputers) which offer good generators and compilers which produce in-line code. On the other hand, even better (but not faster) generators are now available to anyone in Fortran, offering in addition portability, which no manufacturer offers.

2. Pseudorandom numbers

These are the general-purpose random numbers traditionally used for most Monte Carlo calculations.

2.1. Testing good distributions

Zaremba [1] has pointed out that

As far as pseudorandom numbers are concerned, the traditional term "tests of randomness" is a misnomer. Surely, in contrast to their name, the object of such tests is not the random origin of the sequences, since this would amount to testing a hypothesis *known* to be false.

Indeed, pseudorandom numbers are not truly random, and it turns out to be very difficult if not impossible to make a mathematically rigorous definition of pseudorandomness (See, for example, Knuth [2] who has a good discussion of this philosophical point).

Nevertheless, we somehow have to express the fact that pseudorandom numbers should appear to be truly random, even if they are not, and for want of a better word, we shall call this property "randomness", not to be confused with the more usual definition, used for example by Zaremba, which we

distinguish by the name "true randomness". More concretely, we take "randomness" in this sense to mean that a sequence of pseudorandom numbers should have the same probability of passing a "statistical test" as truly random numbers would have. (Not better!) A statistical test may be based on the value of any function of the sequence of pseudorandom numbers. It is sufficient that the expected distribution of that value be known (or calculable numerically) for a truly random distribution, then by considering the value of the function for the given pseudorandom sequence, compared with the known expected distribution of that value for truly random numbers, one obtains a confidence level for the test. If many tests are applied and the confidence levels are calculated correctly, and if the tests are independent, the confidence levels should be uniformly distributed between zero and one if the pseudorandom generator is "good". The formal difficulty arises mostly from the fact that the number of possible tests is uncountably infinite, and in addition they are of course not all independent.

Over the years, considerable experience has indicated what kinds of tests are likely to find the weaknesses of typical generators, and modern tests are much more stringent than most of the older ones. Modern generators are expected to pass all the old tests as well as those tests which traditional generators are known to fail. Probably the most extensive presentation of pseudorandom number testing is given by Knuth [2], but should be updated by the more severe tests giving in ref. [3]. A good example of how to apply such tests systematically is ref. [4]. Random number testing will not be further discussed here, except to mention that any pseudorandom generator likely to have a "lattice structure" (see below) should be subjected to the "spectral test", a simple example of which is given in ref. [5].

2.2. *Pseudorandom generation methods (simple generators)*

We define a simple generator as one for which the maximum period is limited by the number of states that can be represented in one computer word (where a computer word is defined as the entity upon which the local computer likes to perform its integer arithmetic). Thus, as mentioned above, for the popular 32 bit computers, simple generators are limited to a period of about 10^9. The general purpose generators recommended below combine two or more simple generators to attain a longer period and better distribution.

In recent years, three classes of simple generators have been used most extensively. These are generally known as *multiplicative linear congruential (MLCG), Fibonacci*, and *shift register* (also known as Tausworthe) generators. They all have severe weaknesses, but whenever it is known how to get around them, the weaknesses are known as "theoretical understanding".

2.2.1. *MLCG*

The multiplicative linear congruential generator, first used in 1948 by D.H. Lehmer, is one of the oldest, and probably still the best simple generator, even though it has a well understood weakness. In the basic MLCG method, each successive integer is obtained by multiplying the previous one by a well chosen multiplier, optionally adding another constant, and throwing away the most significant digits of the result:

$$s_{i+1} = (as_i + c) \bmod m$$

where a is the well chosen multiplier and m is usually equal to or slightly smaller than the largest integer that can be represented in one computer word. The constant c can be chosen equal to zero, which simplifies the method somewhat and produces sequences about as good as any other, but then an exact zero cannot be generated, and the choice of both a and m may depend on c. The integer seeds s_i must be converted to floating-point numbers in the range (0,1) by dividing by m.

Marsaglia [6] discovered and explained the basic weakness of this method: If d-tuples of such numbers are used to represent points in d-dimensional space, then these points have a lattice structure, that is they all lie on a certain number of hyperplanes, far less than the largest possible number which would be

expected of a truly random sequence. Since it has been known for a long time how to get sequences of maximum period (the necessary and sufficient conditions are given, for example, in ref. [2]), the search for good multipliers is essentially reduced to maximizing the number of hyperplanes, for simple MLCG generators. The so-called "spectral test" [5] is the best way of analyzing the hyperplane structure of any MLCG.

The basic MLCG method should be portable when written in a language like Fortran, but in its simplest expression it requires multiplying rather long integers, and many implementations make use of the convenient peculiarity of many arithmetic units which simply throw away the most significant digits when an integer overflow occurs. Such code is of course not portable, but l'Ecuyer [4] shows how to write this in a guaranteed portable way, even when m is not a power of two.

Some famous multipliers

Even though, in the words of Niederreiter [7], "*There is no such thing as a universally optimal multiplier*", we list here a few of the better known multipliers which have been used in simple MLCGs, with comments as to how good they are now generally considered to be. It should be borne in mind that all the methods given below, except (7), are limited to a period of less than $\approx 10^9$, so none are good enough for long calculations. Except where noted, all the multipliers given below are used with $c = 0$.

1. $a = 23$, $m = 10^8 + 1$: This is the original formula used by Lehmer in 1948, and is not very good by today's standards, although the higher order correlations are not as bad as for the following generator. The constants were chosen mainly to exploit hardware peculiarities, which was important in those days.

2. $a = 65539$, $m = 2^{29}$: This is the infamous RANDU, supplied by IBM in the early days of their 360 series, and was based on a theoretical expression which showed that this multiplier should produce the smallest possible serial correlations. Unfortunately, it turns out to have catastrophic higher-order correlations, which many users have observed. We now know that any multiplier congruent to 5 mod 8 (that is, whose binary representation ends in ...101) would have been better, but this was not known at the time.

3. $a = 69069$, $m = 2^{32}$: This popular multiplier has been used in many generators, probably because it was strongly recommended in 1972 by Marsaglia. In particular, it is the multiplier in RN32, a generator proposed by James [8] for applications which must be portable but do not require very long sequences. It is also the multiplier for the Vax generator MTH$RANDOM, where it is used with $c = 1$. It is in fact quite good for such a small multiplier, its main weakness showing up only in 6 or more dimensions, but according to some criteria, it is far from optimal (see, for example ref. [2], p. 104, and ref. [7], p. 1026). Its best property is that it is easy to remember.

4. $a = 7^5 = 16807$, $m = 2^{31} - 1$: This generator was also developed by IBM for its system/360, where it is known as SURAND. It was proposed on the basis of the state of the art in 1968, just after the discovery of hyperplanes by Marsaglia, and although acceptable for most calculations, it is surely not the best multiplier, probably not even as good as 69069.

5. $a = 1664525$, $m = 2^{32}$: This is the best multiplier for m $= 2^{32}$, according to the criteria of Knuth [2]. It is used in the INMOS Transputer Development System (IMS D700D).

6. $a = 742938285$, $m = 2^{31} - 1$: According to the criteria of l'Ecuyer [4], this is the best simple algorithm, but is not easily made portable. His best "portable" constants (see RANECU below) are $a = 40014$, $m = 2147483563$, and $a = 40692$, $m = 2147483399$.

7. $a = 5^{15}$, $m = 2^{47}$: This is a traditional CDC generator, making use of the 48-bit integer arithmetic used on their 60-bit machines. The long word gives a long period ($\approx 10^{13}$) and very good distribution. Do not trust the low-order bits however.

2.2.2. Fibonacci

A Fibonacci series is one in which each element is the sum of the two preceding elements. A Fibonacci random number generator is a generalization in which each number is computed by performing some operation on the two preceding numbers, the usual operations being addition, subtraction, and the "exclusive-or" operation. Since simple Fibonacci generators are not very good, one always uses *lagged* Fibonacci sequences, in which each number is the result of an arithmetic or logical operation between two numbers which have occurred somewhere earlier in the sequence, not necessarily the last two:

$$s_i = (s_{i-p} \odot s_{i-q}) \bmod m,$$

where $\odot$ is some binary or logical operation, and p and q are the lags, defined such that $p > q$. There is very little theory about the distributions of such pseudorandom sequences, but one knows how to calculate the period, and it is possible to generate quite long sequences in this way. Marsaglia gives a good reason why $\odot$ should be ordinary addition or subtraction, but not exclusive-or, because under the exclusive-or operation, a bit in a given position in the result depends only on the two bits in the same position in the two operands, whereas addition and subtraction produce some actual mixing of the bits.

The basic theoretical result concerning generators of this type is that if p and q are chosen among the set given ref. [2], then the period is $(2^p - 1)(2^{m-1})$. One interesting feature of Fibonacci generators is that they can work (even portably) directly with floating-point representations of numbers, without the need to convert integers. Examples are given below.

In spite of the extensive lack of theoretical understanding, Fibonacci generators are the basis of the generators recommended by both Knuth [2] and Marsaglia [9]. The latter is RANMAR, described and recommended below.

2.2.3. Shift register or tausworthe

This class of generators is based on the same formula as lagged Fibonacci generators, but with $m = 2$, so that only individual bits are generated, which are then collected into words, making use of a shift register, whence the name. The operator $\odot$ is invariably the exclusive-or, which leaves these generators open to the copious criticism of Marsaglia [3,10], who gives examples of bad generators of this type. They have, however, been the subject of considerable interest, and there seems to be no proof that a good generator cannot be based on this method. In particular, people at IBM [11,12] have produced such generators which although not portable, are very fast, have an arbitrarily long period, and have so far performed well in tests.

2.3. Improving simple generators

2.3.1. General techniques

The need for pseudorandom number generators of useful period greater than 10^9 is obvious from the fact that there are now several models of widely available computers which can generate that many numbers in a few minutes. If we assume that computer cycle times will not descend much below a nanosecond in the coming years, it is reasonable to aim for good generators with useful periods of the order of the number of nanoseconds in a year, or about $\pi \times 10^{16}$. Since $2^{60} \approx 10^{18}$ is somewhat larger than this limit, it should be sufficient to combine two 32-bit generators if it can be done in such a way as to get the equivalent of a 64-bit generator.

It is not entirely obvious how to improve a bad generator. Probably the most common mistake is to think that correlations like the Marsaglia effect in MLCG are due only to the "regular" way that the random numbers are used (a fixed number of them per loop), and so things should improve if we occasionally throw away a few numbers. Of course the primary effect of such a technique is to shorten the period rather than lengthen it, but apart from this, such a technique can be expected to improve the randomness only if the decision when to throw away a number is not based on the generator itself. This should be obvious since any attempt to use a bad generator to improve itself will introduce modifications which will be correlated with the very defect one is trying to eliminate. The exact outcome may be hard to predict, but one should hardly be surprised if it makes things worse instead of better. Considerations such as these lead to the following guidelines for improving a simple generator:

- Do not throw away or waste random numbers in order to improve a generator. This mainly shortens the period.
- Do not use the simple generator itself in the algorithm to improve it. Introduce some new randomness by using a different (hopefully independent) random number generator, even if it is not very good.

There are two commonly used methods for improving a generator by using a second generator, where one can show that the new generator will be more random than the original provided the two generators are independent.

- *Shuffling* uses the second generator to choose a random order for the numbers produced by the first generator. This is done by first filling a buffer of a given size from the first generator, then using the second to choose one of these numbers, replacing it by the next in the first sequence. Such a technique is particularly useful for quasirandom numbers, where the original sequence has some property such as equi-distribution which one wants to preserve globally but disturb locally. It is not very efficient for pseudorandom numbers since it does not use much of the randomness of the second sequence, all of the actual numbers coming from the first.
- *Bit-mixing* actually combines the numbers in the two sequences using some logical or arithmetic operation. If the two original sequences are denoted by s and t, then the new sequence r is given by $r_i = s_i \odot t_i$, where $\odot$ was traditionally always the exclusive-or operation, but for reasons given above in connection with Fibonacci generators, ordinary addition or subtraction, modulo one, is now preferred.

2.3.2. Extensions to particular algorithms

In addition to the general techniques mentioned above, most simple generators allow intrinsic extensions which will improve their distributions at the expense of added computation time and space. For example, the popular MLCG generator can be extended from

$$s_{i+1} = (as_i + c) \bmod m$$

to

$$s_{i+1} = (as_i + bs_{i-1} + c) \bmod m$$

which is known to increase the maximum number of Marsaglia hyperplanes to the number which would be obtained for the simpler formula in half the number of dimensions. To use the second formula requires carefully choosing two multipliers a and b. If this is still not good enough, further improvement can be obtained by adding still more terms to the formula and finding additional good multipliers.

2.3.3. Very long period (VLP) methods

Techniques of the type described above necessarily involve an increase of computation time in order to lengthen the period, with the time increasing like the log of the period for long periods. This is acceptable for attaining modest improvements, but a new class of algorithms is needed for very long periods.

The general technique for VLP generators (for example, RANMAR and ACARRY, described below) is as follows:

- An internal table is set up, containing a large number of seeds (typically between ten and a few hundred), and the values of a few indices (typically two) pointing to seeds in the table, are also initialized.
- A pseudorandom number is generated by combining only those seeds corresponding to the current values of the indices.
- The seeds just used are updated as in simple methods, and the indices are incremented to point to other seeds.

The amount of computation involved in such a method is only about twice as much as for the very simplest methods, but the period is now limited not by the number of states representable in one or two words, but in the entire table, which can be made as large as necessary. For simplicity of use, the original initialization of the seed table is usually based on a single integer supplied by the user, which starts a simple MLCG generator. On the other hand, if it may be required to restart the generation from any arbitrary point, then the full seed table at that point must be saved, along with the index values.

3. Acceptable pseudorandom generators

3.1. The McGill generator super-duper

In the mid 1970s the need for generators of longer period for 32-bit computers was already becoming apparent, and this gave rise to the first widely distributed combined generator, written by Marsaglia and co-workers at McGill University. It combines a MLCG and a shift-register generator, and is written in IBM assembler, so is not portable, but is now used at many IBM sites. In the CERN Program Library it is known as RNDM2. It is packaged in the traditional way, as a function returning one random number at a time, and is initialized by two 32-bit integer seeds which must be chosen with care, so it is not easy to generate simultaneous independent sequences.

This is by far the best of any of the traditional generators commonly used on IBM mainframes today, but Marsaglia now suggests that in the assembler code, the "exclusive-or" operation for combining the generators be replaced by a simple "add" instruction. Even then it will not be quite as good as those described below, and is of course not portable.

3.2. RANECU: the algorithm of l'Ecuyer

L'Ecuyer [4] describes the current state of the art in MLCG methods and shows how to make portable generators by doing integer arithmetic in such a way that both operands and results are guaranteed to stay within the range of 32-bit (or 16-bit) computers. He finally recommends a method which combines two simple MLCG's for 32-bit computers, and three simple MLCGs for 16-bit machines. The periods are $\approx 2 \times 10^{18}$ and $\approx 10^{12}$, respectively.

For maximum period, the different values of m in one generator must be the largest relatively prime numbers which can fit in one word, and for portability the multipliers must be less than $\sqrt{m}$. L'Ecuyer [4] shows how to find the best constants satisfying these constraints and gives the actual Pascal code to implement the mehtods. (The fact that they are given in Pascal should not be taken as an indication that they are intended only for toy applications. A Fortran adaptation is given here.)

By traditional standards, this generator is relatively slow, the 32-bit version requiring for each number generated two integer divisions, six integer multiplications, and several additions, of which three are conditional. However, on all machines I have tried, the pure calculation is comparable with the time to set

up the function call alone, so if implemented such that each call returns a vector of random numbers, it becomes about as fast as traditional hand-coded algorithms (some timings are given below for those interested).

I see only two very minor drawbacks to this method. The least important is that the numbers generated on different machines are not bit-identical, but are only equal to the accuracy of the machine arithmetic. This is because they are normalized by multiplying by the inverse of a very large odd integer, and this inverse cannot be represented exactly in binary. The other small problem is that it is not very convenient for generating long disjoint subsets, although l'Ecuyer does indicate how it can be done by calculating values of a^i for large and well-separated values of i. He gives a reference where some values are tabulated.

The following Fortran code is the generator recommended by l'Ecuyer for 32-bit machines, translated from Pascal and adapted to generate an array of numbers in one call:

```
      SUBROUTINE RANECU (RVEC,LEN)
C           Portable random number generator proposed by l'Ecuyer
C                in Commun. ACM 31 (1988) 743
C           slightly modified by F. James, 1988, to generate a vector
C           of pseudorandom numbers RVEC of length LEN
      DIMENSION RVEC( * )
      SAVE ISEED1,ISEED2
      DATA ISEED1,ISEED2 / 12345, 67890 /
C
      DO 100 I =  1, LEN
      K = ISEED1/53668
      ISEED1 = 40014 * (ISEED1 - K * 53668) - K * 12211
      IF (ISEED1 .LT. 0) ISEED1 = ISEED1 + 2147483563
C
      K = ISEED2/52774
      ISEED2 = 40692 * (ISEED2 - K * 52774) - K * 3791
      IF (ISEED2 .LT. 0) ISEED2 = ISEED2 + 2147483399
C
      IZ = ISEED1 - ISEED2
      IF (IZ .LT. 1) IZ = IZ + 2147483562
C
      RVEC(I) = REAL(IZ) * 4.656613E-10
  100 CONTINUE
      RETURN
C
      ENTRY RECUIN(IS1,IS2)
      ISEED1 = IS1
      ISEED2 = IS2
      RETURN
C
      ENTRY RECUUT(IS1,IS2)
      IS1 = ISEED1
      IS2 = ISEED2
      RETURN
      END
```

L'Ecuyer gives convincing evidence that the numbers generated are very well distributed.

3.3. RANMAR: the algorithm of Marsaglia, Zaman and Tsang

This generator [9], is the first of a new generation of portable VLP (very long period) methods. It has a period of $2^{144} \approx 2 \times 10^{43}$, is completely portable, giving bit-identical results on all machines with at least 24-bit mantissas in the floating-point representation (i.e. all the common computers of 32 bits or more). It satisfies very stringent tests, even though the only precise theoretical understanding is the knowledge of the period. It is fast enough (somewhat faster than RANECU), largely because it works internally in floating-point representation, rendered portable by clever coding techniques which are somewhat unusual, but perfectly well-defined Fortran.

A most exceptional property of this generator is the extreme simplicity of generating independently disjoint sequences. The generator must be initialized by giving one 32-bit integer (in the original version, four smaller integers), *each value of which gives rise to an independent sequence of sufficient length for an entire calculation*. This means that in a collaboration between different physicists, each physicist can be assigned one number between zero and 9999 as the last four decimal digits of the initiator, and he will be assured of not overlapping the sequences of any other, even though he still has about 90 000 possibilities for the other digits at his disposal for independent initialization. That is, the program can generate about 900 million different subsequences, each one very long (average length $\approx 10^{30}$).

On the other hand, there is a small price to pay for the exceptionally long period: The complete specification of the state of the generator at a given point (for example to be able to regenerate a given event in the middle of a calculation) requires one hundred and two full words (the contents of the COMMON block RASET1 in the listings below). This contrasts with the one word (two words) necessary for a MLCG of period $\approx 10^9$ ($\approx 10^{18}$).

The algorithm is a combination of a Fibonacci sequence (with lags of 97 and 33, and operation "subtraction plus one, modulo one") and an "arithmetic sequence" (using subtraction). The "arithmetic sequence" has not yet been mentioned here, because it is of little interest by itself, and is not used in many standard generators, but is claimed to be good enough when combined with another method, like the lagged Fibonacci, which is already almost good enough. The combining of the two sequences is again done with the operation "subtraction plus one, modulo one", and all operations are carried out in floating-point assuming at least 24-bit mantissas. The starting table of 97 values is initialized using a combination of a lagged Fibonacci method using three lags, and a MLCG using $a = 53$, $m = 169$.

The Fortran code given below is essentially that given by Marsaglia and Zaman [9], except that the present version returns a vector of numbers rather than just one.

```
      SUBROUTINE RANMAR (RVEC,LEN)
C  Universal random number generator proposed by Marsaglia and Zaman
C  in report FSU-SCRI-87-50
C          slightly modified by F. James, 1988, to generate a vector
C          of pseudorandom numbers RVEC of length LEN
C          and making the COMMON block include everything needed to
C          specify completely the state of the generator.
      DIMENSION RVEC( * )
      COMMON/RASET1/U(97),C,CD,CM,I97,J97
C
      DO 100 IVEC = 1, LEN
      UNI = U(I97)-U(J97)
      IF (UNI .LT. 0.) UNI = UNI + 1.
      U(I97) = UNI
      I97 = I97-1
```

```
      IF (I97 .EQ. 0) I97 = 97
      J97 = J97-1
      IF (J97 .EQ. 0) J97 = 97
      C = C-CD
      IF (C .LT. 0.) C = C + CM
      UNI = UNI-C
      IF (UNI .LT. 0.) UNI = UNI + 1.
      RVEC(IVEC) = UNI
  100 CONTINUE
      RETURN
      END
```

The initialization routine given below is also slightly modified from the original version given in ref. [9]. (The original form of RSTART as it appeared in FSU-SCRI-87-50 required four small integers for initialisation; the adaptation given here and renamed RMARIN simplifies somewhat the initialization, accepting one integer between zero and 900 000 000 rather than four smaller integers.)

Some implementation hints: In the form given here, RANMAR cannot know if RMARIN was in fact called to perform the necessary initialization. In order to remain strictly portable, this can be fixed only if RMARIN is incorporated as an entry point into RANMAR, in which case RANMAR could recognize (through the use of a variable set in a DATA statement) when the user had failed to initialize it, and in that case perform an initialization with some default seeds, thereby making the generator more convenient for general users. The implementation of entry points is standard Fortran 77 and portable. Note also that if RMARIN is incorporated as an entry point in RANMAR, the COMMON block is no longer needed, but if it is removed, those variables must be declared in a SAVE statement. Other possible implementation options include providing entry points for inputting and outputting the entire seed table for restarting, or a simpler (but longer) restarting procedure based on counting the number of numbers generated since the last call to RMARIN. All the above features are incorporated into the code PSEUDORAN distributed by the CPC Program Library, but only the basic ideas of the initialization are illustrated by the following code listing:

```
      SUBROUTINE RMARIN(IJKL)
C           Initializing routine for RANMAR, must be called before
C           generating any pseudorandom numbers with RANMAR. The input
C           value should be in the range: 0 < = IJKL < = 900 000 000
      COMMON/RASET1/U(97),C,CD,CM,I97,J97
C  This shows correspondence between the simplified input seed IJKL
C       and the original Marsaglia-Zaman seeds I,J,K,L
C  To get standard values in Marsaglia-Zaman paper,
C           (I = 12, J = 34, K = 56, L = 78) put IJKL = 54217137
      IJ = IJKL/30082
      KL = IJKL - 30082 * IJ
      I = MOD(IJ/177, 177) + 2
      J = MOD(IJ, 177) + 2
      K = MOD(KL/169, 178) + 1
      L = MOD(KL, 169)
      PRINT '(A,I15,4I4)',' RANMAR INITIALIZED: ', IJKL,I,J,K,L
      DO 2 II = 1, 97
      S = 0.
```

```
      T = .5
      DO 3 JJ = 1, 24
           M = MOD(MCD(I * J,179) * K, 179)
           I = J
           J = K
           K = M
           L = MOD(53 * L + 1, 169)
           IF (MOD(L * M,64) .GE. 32) S = S + T
    3 T = 0.5 * T
    2 U(II) = S
      C = 362436./16777216.
      CD = 7654321./16777216.
      CM = 16777213./16777216.
      I97 = 97
      J97 = 33
      RETURN
      END
```

3.4. *ACARRY: the algorithm of Marsaglia and Zaman*

The most recent effort of Marsaglia and friends [13,14] is a whole class of VLP generators known as *add-and-carry*. I will refer to this class of generators generically as ACARRY, but the particular variation which I propose here is actually *subtract-and-borrow*, and the name of the subroutine I propose is RCARRY. The algorithm looks very much like lagged Fibonacci, but with the occasional addition of an extra bit, according to whether or not the Fibonacci sum was greater than one. The basic formula is:

$$x_n = (x_{n-r} \pm x_{n-s} \pm c) \bmod b$$

where $r > s$ are the lags, and c is a carry bit, equal to zero unless the sum was greater than b, in which case it is a one in the least significant bit position. The word size b can be chosen = 2 in which case it generates random bits, or a larger number (usually a power of two) for generating longer numbers. In the example proposed below, we take $b = 2^{24}$ to generate 32-bit floating-point numbers with 24-bit mantissas.

As with the Fibonacci method, r and s are chosen from a set of magic numbers for which the method is known to yield a very long period. And this algorithm also requires storing the previous r seeds. For $b = 2^{24}$, a convenient choice is $r = 24$, $s = 10$, which gives a period only a factor of 48 smaller than the number of different states that can be represented by 24 24-bit numbers, which is $(2^{24})^{24}$. That is, the period for the generator given below is $\approx 2^{570}$ or $\approx 10^{171}$. The state of the generator at any time can be specified by the values of 24 24-bit integers plus two small integers and the value of the carry bit, which can easily be packed into a 25th word.

Only the generator proper is shown below. In addition, it is necessary to initialize the vector of 24 floating-point seeds as well as the two indices I24 and J24, as in RMARIN, and the starting value of CARRY must also be initialized, but it can be started with zero. The code PSEUDORAN distributed by the CPC Program Library performs also the initialization and offers entry points for inputting and outputting the state of the generator at a given time. The normal initialization is either by default (which will of course always yield the same sequence) or by inputting an integer (24 bits or less) each value of which starts a very long ($\approx 10^{160}$) subsequence which will not overlap with any other.

```
      SUBROUTINE RCARRY (RVEC,LENV)
C               Portable Pseudorandom Number Generator with period of
C               about (1/48) * (2 * * 24) * * 24 = 2 * * 570 = 10 * * 171
```

```
C          author: F. James, CERN, 1989
C          algorithm due to: G. Marsaglia and A. Zaman
C
      DIMENSION RVEC(LENV)
      DIMENSION SEEDS(24)
      PARAMETER (TWOP24 = 16777216.)
      PARAMETER (TWOM24 = 1./TWOP24)
C               the basic generator algorithm only
      DO 100 IVEC = 1, LENV
      UNI = SEEDS(I24) - SEEDS(J24) - CARRY
      IF (UNI .LT. 0.) THEN
          UNI = UNI + 1.0
          CARRY = TWOM24
      ELSE
          CARRY = 0.
      ENDIF
      SEEDS(I24) = UNI
      I24 = I24 - 1
      IF (I24 .EQ. 0) I24 = 24
      J24 = J24 - 1
      IF (J24 .EQ. 0) J24 = 24
      RVEC(IVEC) = UNI
  100 CONTINUE
      RETURN
      END
```

4. Conclusions

4.1. Timing

The times given in table 1 are for Fortran loops calling the indicated generator alone. For RANECU, RANMAR and RCARRY, two times are given: one for one call returning 1000 numbers, and the other for 1000 calls returning one number each. The other generators can only return one number per call, so the

Table 1
Pseudorandom number generation time

Generator	$\log_2$ period	Time in μs per random number				
		Cray X-MP	IBM 3090	Vax 8800	Apollo 10000	Apollo 3500
RNDM	30	2.0	0.7	7.0	2.4	30.0
RN32	30	2.0	2.6	5.0	2.4	42.0
MTH$RANDOM	30	–	–	5.0	–	–
RANF	46	0.07	–	–	–	–
Super-Duper	60	–	1.3	–	–	–
RANECU (1000/call)	60	2.5	2.6	9.0	5.5	71.0
RANECU (1/call)	60	4.1	4.5	16.0	7.0	84.0
RANMAR (1000/call)	144	1.1	1.2	5.0	3.4	130.0
RANMAR (1/call)	144	4.5	3.4	12.0	9.1	148.0
RCARRY (1000/call)	568	0.8	0.8	3.0	2.0	46.0
RCARRY (1/call)	568	2.7	3.1	15.0	4.8	65.0

Table 2
Properties of some Pseudorandom Number Generators

Generator	Randomness	Portability	Approx. period	Needed to initialize [wd]	Needed to restart [wd]	Disjoint sequences, no. × length
traditional	unreliable	poor	10^9	1	1	sequential
super-duper	acceptable	none	10^{18}	2	2	sequential
RANECU	good	good	10^{18}	2	2	$(10^9 \times 10^9)$ [a]
RANMAR	good	good	10^{43}	1	100	$10^9 \times 10^{34}$
RCARRY	good	good	10^{170}	1	25	$10^9 \times 10^{161}$

[a] RANECU can make independent subsequences, but not conveniently.

times are given for 1000 calls. The times given are not accurate (or even repeatable) to more than about 10%. Super-duper is in IBM assembler only. The implementation of RNDM (CERN Program Library) is very computer-dependent, but times are given to allow users to compare with a generator which they may already be using. RN32 [8] is a nearly portable Fortran function generating the same numbers on different machines.

The Fortran code for RANECU, RANMAR, and RCARRY was absolutely identical on all machines, and no attempt was made to optimize the code for any particular computer (on some computers, considerable improvement in speed can be attained by customizing the code, but our goal here is rather portability). It can be seen that in most cases there is not much difference in timing from one generator to another, the major exception being RANF which is exceptionally fast because the Fortran compiler produces in-line object code. The better generators are not much slower (and in some cases quite a bit faster!) than the mediocre ones, and a factor of two or three can usually be gained by returning several numbers in one call.

4.2. Summary of basic properties

The most important properties of both traditional and newer recommended pseudorandom generators can be summarized in table 2. All generators produce 32-bit floating-point numbers uniformly distributed between zero and one. The unit "1 wd" means one 32-bit word. The figures given as powers of ten are only approximate.

4.3. Other considerations

4.3.1. Higher precision or shorter word length

The particular algorithms given here are for computers with word length of 32 or more bits, where 32-bit precision is assumed to be sufficient. If higher precision is required (this should be very rare), then all of the algorithms must be extended to produce longer numbers. For the MLCG, which works internally with integer arithmetic, one must not only find multipliers appropriate for larger bases, but also use tricks like the one shown here to do extra-length integer arithmetic and floating-point conversion. Algorithms based directly on floating-point arithmetic are simpler to extend, since double precision floating-point arithmetic is standard in Fortran.

For 16-bit computers, l'Ecuyer [4] gives a good algorithm based on three seeds.

4.3.2. Exact zeros

The routines RANMAR and RCARRY produce floating-point numbers in the range from zero to one, excluding one, but including exact zeros. Although zeros occur on average only once per 2^{24} numbers, they

may be highly bothersome, especially if the subsequent calculations involve for example taking the logarithm of the random number. In addition, the results may be sensitive to the fact that no number will be generated between zero and 2^{-24}. Both these problems may be solved elegantly by the following technique.

- Just before the statement RVEC(IVEC) = UNI, include the following code:

```
      IF (UNI .EQ. 0.) THEN
         UNI = U(J97) * 2 * * - 24          ! (for RANMAR)
or       UNI = SEEDS(I24) * TWOM24          ! (for ACARRY)
         IF (UNI .EQ. 0.) UNI = 2 * * - 48
      ENDIF
```

This produces a uniform distribution of numbers between 2^{-48} and 2^{-24}, with no numbers smaller than 2^{-48}.

4.4. Recommendations

1. Old-fashioned generators like RNDM, RN32, MTH$RANDOM, SURAND, and RANDU should be archived and made available only upon special request for historical purposes or in situations where the user really wants a bad generator. The user who is not sure what he needs should not by default get a generator known to be deficient.

2. The standard form for a pseudorandom number generator should be a subroutine returning an array of random numbers rather than a function returning one number. This is mainly for efficiency, but is also good Fortran programming, since random number generators always have side effects and are not therefore true Fortran functions.

3. RANECU, RANMAR and ACARRY should be available at all computer centres, and the "default" generator, where one is provided, should be one of these. That is, even users who persist in an old calling sequence such as RR = RNDM(DUMMY) should now get numbers produced by a modern generator unless they invoke a special library which may provide historical numbers.

4. If computer time for random number generation is critical, use a generator coded or compiled in-line.

References

[1] S.K. Zaremba, SIAM Rev., 10 (1968) 303.
[2] D.E. Knuth, Semi-numerical algorithms, vol. 2 in: The Art of Computer Programming, 2nd ed. (Addison–Wesley, Redding, MA, 1981).
[3] G. Marsaglia, A current view of random number generators, in: Computer Science and Statistics: The Interface, L. Billard, ed. (Elsevier, Amsterdam, 1985).
[4] P. l'Ecuyer, Commun. ACM 31 (1988) 742.
[5] G. Marsaglia, The structure of linear congruential sequences, in: Applications of Number Theory to Numerical Analysis (Academic, New York, 1972).
[6] G. Marsaglia, Proc. Nat. Acad. Sci. WA 61 (1968) 25.
[7] H. Niederreiter, Bull. Am. Math. Soc. 84 (1978) 957.
[8] F. James, Rep. Prog. Phys. 43 (1980) 1145.
[9] G. Marsaglia, A. Zaman and W.-W. Tsang, Stat. Prob. Lett. 9 (1990) 35.
[10] G. Marsaglia and L.-H. Tsay, Linear Algebra Appl. 67 (1985) 147.
[11] S. Kirkpatrick and E.P. Stoll, J. Comput. Phys. 40 (1981) 517.
[12] Private communication (1988).
[13] G. Marsaglia and A. Zaman, SIAM J. Sci. Stat. Comput., to be published.
[14] G. Marsaglia, B. Narasimhan and A. Zaman, Comput. Phys. Commun. 60 (1990) 345, this issue.